FEATURE ARTICLE

Markus Levy

Interfacing Microsoft's Flash File System

Using Flash Memory Under MS-DOS

Back in issue 18, I discussed the design for a flash memory array based upon a PC/AT add-in board employing SIMMs or IC memory cards. Now we are ready to turn your flash memory platform into a DOS-compatible solid-state disk. In this article I'll show you how to interface Microsoft's special Flash File System (FFS) so you can store and retrieve files on this non-volatile solid-state disk.

Before discussing the structural elements and benefits of the FFS, let's review some basic standard MS-DOS concepts. This will provide the motivation for implementing a special file system for flash memory.

A LOOK AT STANDARD MS-DOS

At the highest level, applications make requests to the MS-DOS function dispatcher through interrupt 21h. Arguments identifying the service desired and its options are typically passed in registers. To relieve application programs of the necessity of managing disk storage space, MS-DOS provides a file system manager. This series of MS-DOS services keeps track of disk storage using file and directory structures.

All block-device accesses by MS-DOS are made through a standardized device driver. MS-DOS makes requests to a block device by passing a data structure, called a request header, to the device driver. The request header contains the desired command and required arguments and is located by the device driver using a special routine, typically named "Strategy." For a read or write request, the logical sector to start the access and the number of logical sectors to transfer are provided. The device driver performs translations from the requested logical sector to the physical location on the device.

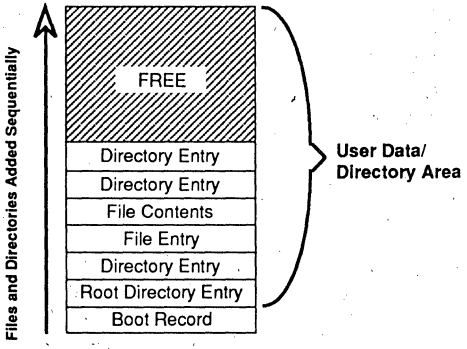


Figure 1—*Flash files are located by following linked-list pointers within each file and directory entry.*

For each block device, MS-DOS maintains four areas: a boot record, a File Allocation Table (FAT), a root directory, and a data area. The boot record is the first section on the disk. It contains a structure known as the BIOS Parameter Block (BPB) supplying MS-DOS with information about the disk, including sector size, sectors per cluster, number of FATs, directory size (number of files), and so on.

MS-DOS requires each disk to have a FAT to keep track of sector/cluster allocation. The FAT contains information on all the clusters that are allocated, free or unusable. It is an array of cluster pointers and each entry has a one-to-one correspondence with a cluster on the disk.

Grouping sectors to form clusters increases efficiency in terms of the memory required to manage the FAT. A cluster can consist of a different number of sectors. Four sectors (or 2048 bytes) per cluster is typical on a hard disk. The larger the cluster or allocation unit, the more potential of wasting space for files not sized to a multiple of the number of bytes in a cluster. For instance, a 20-byte file stored on a disk with a cluster size of 2048 bytes, wastes 2028 bytes.

Following the FAT is the root directory. Directory entries consist of the file name, attributes, time, date, file size, initial cluster number, and reserved bytes. When allocating space to a file, the initial cluster number is updated to point to the initial cluster number used by the file. The value at that location in the FAT points to the next cluster used by the file or contains an end-of-file marker. Thus, the allocation chain is a forward linked list. When extending a file and another cluster is required, MS-DOS replaces the end-of-file marker with a pointer to the next cluster, which is set to an end- of-file marker.

An increase in file size, or any type of file revision, results in a byte-alterable modification to the disk di-

Order Number: 295067-001

rectory. In addition to file size change, modifications include time of last change, date, attributes, and file renaming.

THE NEED FOR AN ALTERNATIVE FILE SYSTEM

Recall that flash memory is a bulk-erase memory. The FAT and directory structures created for the byte-alterable magnetic disk are not ideal for a flash memory solid-state disk (SSD). Updating a FAT or directory entry requires complete erasure of the flash memory components containing the changing bytes. It is possible to implement a flash memory disk based on this approach, but the write latency times are unacceptable for general use. When a file is added, deleted, or modified, the directory could be copied to a RAM buffer and modified to reflect the change. After the flash memory device that contains this directory is erased, the modified directory is copied back.

Disk imaging is another method of implementing the standard MS-DOS FAT scheme on a flash memory SSD. Using this method, files are first copied to a floppy disk. Then a special utility performs a disk copy transferring the FAT, directory, and all files to the flash memory SSD. This approach is useful for building an application cache that is FAT file system compatible, but all flexibility is lost.

Microsoft has developed a special file system utilizing the attributes of flash memory. To minimize fragmentation losses and allow arbitrary extension of files, the flash memory file system uses variable-sized blocks rather than the sector/cluster method of standard MS-DOS file systems. This flexibility is provided by employing a linked-list structure; that is, chaining files together using address pointers located within directory entries for each file.

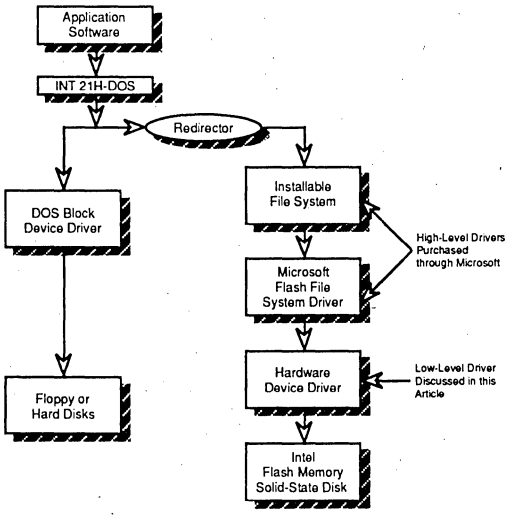


Figure 2—*A two-level architecture provides a consistent application interface while allowing for a variety of flash-memory hardware platforms.*

Files and directories are written to the flash memory SSD using sequentially free memory locations—a stack-like operation (Figure 1). When the "stack" is full, the desired files are copied to another disk and the current disk is erased for reuse.

File and subdirectory information is attached to the beginning of each file, unlike the standard MS-DOS approach of directory and FAT placement. As directory and file entries are added, they are located by building a linked list. Besides containing the standard fields (e.g., name, extension, time, date of creation), a directory or file entry contains a status byte and various pointers used for the linked-list structure. The status byte, besides indicating whether a file/subdirectory exists or is deleted, also signifies valid sibling and/or child pointers and if a directory entry pertains to a file or a directory.

When a directory or file is requested, the flash memory SSD is searched beginning at the head of the linked list. The chain is followed from pointer to pointer until the correct entry is found. If the search arrives at the chain's end (an FNULL identifier is encountered), the system responds analogously to MS-DOS with a "File not found" message.

When deleted versions of a file appear on the flash memory SSD, the file system finds the most recent version. The status byte contains bit fields that indicate whether a particular file is valid or deleted. The directory information of a deleted file is still used for pointers of the linked list and the search proceeds until it finds the most recent and valid version.

FLASH FILE SYSTEM: ARCHITECTURE OVERVIEW

The Flash File System consists of two components: `IFS.SYS` (Installable File System) and `FEFS.SYS` (Flash EPROM File System). When an application accesses a disk through interrupt 21h, the MS-DOS kernel checks the drive letter. If the drive has been declared as a flash memory SSD, `IFS.SYS` intercepts the request through a proprietary interrupt 2Fh redirector interface and passes it to

Character Device Header

```
DW      Block Header            ; Pointer to Next Device Driver
DW      0
DW      1100000000000000B       ; Attributes (CHAR, IOCTL control)
DW      Strategy                ; Offset of Strategy Procedure
DW      Interrupt               ; Offset of Interrupt Procedure
DB      'FIFIDEVE'              ; Device Name Used by FEFS to Locate the LLD
```

Figure 3—*The Flash File System doesn't use a FAT and directory structure, but is uses a character device driver which must contain the proper header.*

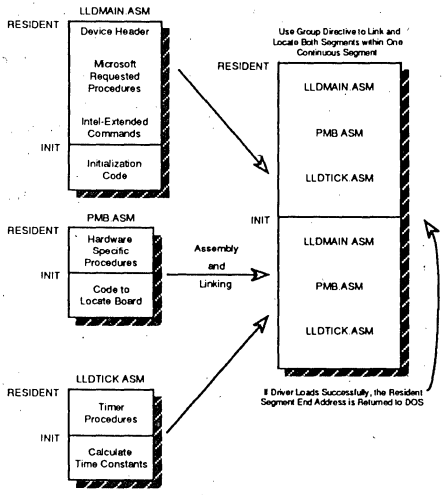


Figure 4—*The INIT portion of the device driver is used only once, at initialization, then "discarded."*

the `FEFS.SYS` driver. `FEFS.SYS` implements the FFS logic, developing and maintaining the linked-list structures.

The Microsoft FFS is implemented as a two-level architecture, where `IFS.SYS` and `FEFS.SYS` represent the high-level driver communicating with a low-level driver (LLD) that is hardware specific (Figure 2). This architecture provides a consistent application program interface while allowing for a variety of flash memory hardware platforms.

The LLD implements a set of device primitives for use by the high-level driver. This is not that different from the FAT file system as we know it. In that environment, MS-DOS implements a high-level, FAT file system driver interacting with a set of device primitives implemented as interrupt 13h.

WRITING THE LOW-LEVEL DRIVER

The remainder of this article will explain how to write this low-level driver providing the bridge between Microsoft's Flash File System and the page-memory board described in the December issue. While I would need forty pages to completely explain all the details of this implementation, there is enough here for you to understand the LLD software that can be obtained from the Circuit Cellar BBS. Also, if you have no intentions of implementing a DOS-compatible SSD, many of the device primitives from the low-level driver can be extracted for any type of flash memory application.

The LLD consists of three components: an MS-DOS device driver, the procedures called by Microsoft's FFS, and the page-memory board hardware-specific procedures. I have written the LLD in several modules to simplify any modifications necessary to accommodate hardware variations. The module `LLDMAIN.ASM` contains the MS-DOS interface routines, the Microsoft-requested procedures, and special Intel-extended procedures. `PMB.ASM` contains procedures specific to the page-memory board hardware, such as setting the page number and turning on V_{PP}. All the functions contained in this module control the I/O functions on the board. `LLDTICK.ASM` contains procedures associated with the timing requirements of the flash memory components, as well as routines used to provide a V_{PP} turn-off delay.

The programming and erase voltage, V_{PP}, is generated on the page-memory board using a DC-DC converter. When this converter is switched on, it takes anywhere from 20 to 100 milliseconds for V_{PP} to arrive at a stable voltage. This time depends on the amount of capacitive loading and the circuit used, as some have faster start times. If you are designing your hardware for a desktop system, V_{PP} can remain switched on. However, in battery-powered systems, V_{PP} should be switched off when not in use to conserve power. To accommodate these applications, I have written a procedure to generate a V_{PP} turn-off delay. This is similar to that for a floppy disk in that after two seconds of nonuse, V_{PP} is switched off. If several blocks of data are being written consecutively, your software will not have to delay waiting for V_{PP} to ramp up every time a new block is written.

The V_{PP} turn-off delay is calculated by installing an interrupt 1Ch (time-of-day clock) filter. This interrupt is generated every 18.2 milliseconds. Before servicing the original interrupt, our filter increments a count value. When that count value reaches 36, the procedure to turn V_{PP} off is called.

Since the LLD is an installable device driver, it requires a standard

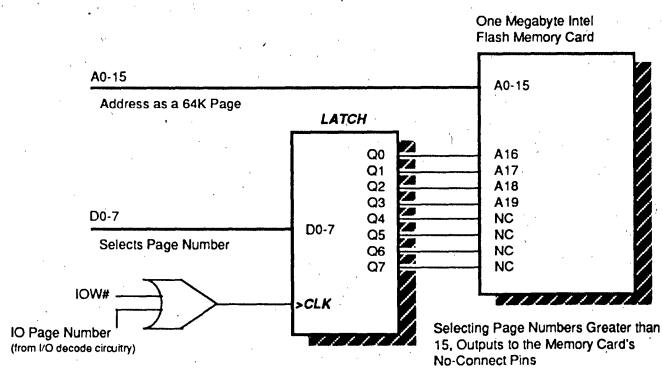


Figure 5—*Device density is dynamically determined, with page numbers beyond the valid size of the device handled by page wrapping.*

MS-DOS–compatible interface that provides a request header and entry points into the Strategy and Interrupt procedures. This portion of code primarily performs the initialization of the device driver.

Unlike the magnetic disk, the FFS does not implement a FAT and directory structure with sectors. Therefore, it is designed as a character device driver rather than a block device driver and must contain a character device driver header (Figure 3). During installation of the character device, MS-DOS passes a command number in a request header to the Interrupt procedure which dispatches a call to the `InitDriver` routine (command zero). This procedure is performed only once, immediately after the device driver is loaded into memory. The initialization procedure of the character device, `InitDriver`, is primarily responsible for locating the page-memory board, computing the time constants for the flash memory program and erase algorithms, and determining the quantity of flash memory available.

INITIALIZATION

Recall from the page-memory board design that the first four I/O ports are read to obtain the board's signature. A procedure, `SearchPMB`, called by `InitDriver`, reads sequential I/O ports until finding this signature. When the signature is found, the base port address is stored in a memory location to be used for future I/O port access. If the board is not present, the driver aborts its installation and returns the system memory to MS-DOS. This is done by passing an offset of zero back to MS-DOS in the INIT request header. Each of the three modules (`LLDMAIN.ASM`, `PMB.ASM`, `LLDTICK.ASM`) are divided into a `RESIDENT` and an `INIT` segment. These segments are joined using the group directive to ensure they are linked and loaded consecutively in memory. Using this technique, if the page-memory board is found, the ending offset of the resident portion of the device driver is passed back to MS-DOS (Figure 4). This "throws

away" the INIT portion which is not needed after initialization.

The page-memory board has four sockets for Intel Flash Memory single in-line memory modules (SIMMs). The hardware functions with one or more sockets populated, so during initialization the available memory must be determined. Similarly, with the Intel Flash Memory Card, an interesting situation is faced in accommodating density variations. This is best understood using the memory card as an example.

One- and four-megabyte card densities are available today, but the page-memory board's card socket handles up to 64 megabytes (based on the Personal Computer Memory Card International Association, PCMCIA, specifications). Assume that a one-megabyte card has been installed (although our software doesn't know this yet) containing eight one-megabit components (28F010). The first step in determining the module's density is reading the flash memory device ID from offset zero of page zero to obtain the device size. Comparison against a table of IDs versus densities allows the calculation of the page number needed to access the next component.

Theoretically, this process would continue, adding up the total number of components and multiplying that number by the component size to calculate the total module density. However, the page-memory board responds to setting the page register beyond the valid page range of the flash memory installed (Figure 5). For example, the one-megabyte memory card accommodates sixteen (numbered 0–15) 64K-byte pages. Pages 0, 16, 32, and so on, access the same memory location because of the wrap-around phenomenon. This inaccurately determines an infinite module size. How does our software know when to stop? Notice in the code (Listing 1) that the first device (page zero) is left in the READ_ID mode. Looping through the rest of memory, a wrap-around is detected when the device ID is already present in the first location of the device. This condition is used to terminate the loop.

Besides initialization, the MS-DOS device driver for FFS supports IOCTL reads and writes. The device driver uses the IOCTL commands to return control information to the program regarding the device. FFS issues the IOCTL commands to get and set the entry point for the low-level driver. Unlike block devices, character devices are located with a file open request. When FEFS.SYS installs, it links into the low-level driver by opening the file FIFIDEV£ (£ is the British Pound Sterling or IBM extended ASCII 9Ch), and performing an IOCTL read to obtain the pointer to the entry of the low-level driver.

Character devices do not support drive letters. You must use a "pseudo-block" device header during driver installation to reserve DOS letters for use by the FFS (Listing 2). After the character device installs, the block-device driver links the device driver into the device driver chain. Drive letters are established by providing fake BPB information in the block device driver header to pass back to MS-DOS.

```
GetPhyChar PROC NEAR
        ASSUME CS:PROG, DS:PROG, ES:NOTHING

        CALL    TurnVppON     ; Turn Vpp ON for READID_CMD
        XOR     AX,AX         ; AX contains page number
        CALL    SetPage       ; Routine to set page

        MOV     ES,FrameSeg   ; Point to page frame segment
        XOR     DI,DI         ; Point to start of device

        MOV     ES:WORD PTR[DI],(READID_CMD SHL 8) OR READID_CMD

        MOV     AX,ES:[DI]    ; Read manufacturer's ID
        CMP     AX,(INTEL_ID SHL 8) OR INTEL_ID ; Intel Devices?
        JE      GPC2          ; Yes, continue
GPC1:
        MOV     ES:WORD PTR[DI], (READ_CMD SHL 8) OR READ_CMD
        STC                   ; Indicate error and exit
        JMP     GPCExit
GPC2:
        MOV     BX,2          ; num of 64K pages/device
                              ;  for 1 Meg devices

; ----- Compute total size of media

        XOR     AX,AX         ; Clear regs for accumulation
        MOV     CX,AX         ; Count num of device pairs
        SHL     BX,1          ; 2 Devices per 16 bits
GPC3:
        INC     CX            ; Count num of device pairs
        ADD     AX,BX         ; Point to next device pair
        CALL    SetPage       ; Select page for next device
                              ; Already in READ_ID mode?
        CMP     ES:WORD PTR [DI],(INTEL_ID SHL 8) OR INTEL_ID
        JNE     GPC4          ; No, Continue
        JMP     GPC5          ; Yes, We're done
GPC4:
        MOV     ES:WORD PTR [DI],(READID_CMD SHL 8) OR READID_CMD
                              ; Intel Devices?
        CMP     ES:WROD PTR [DI],(INTEL_ID SHL 8) OR INTEL_ID
        JNE     GPC5

; Set READ mode and check next device
        MOV     ES:WORD PTR [DI],(READ_CMD SHL 8) OR READ_CMD
        JMP     GPC3
GPC5:
        MOV     ES:WORD PTR [DI],(READ_CMD SHL 9) OR READ_CMD
        MOV     AX,CX         ; Get num of device pairs
        SHL     AX,1          ; Compute number of devices
        MOV     BX,2          ; Multiply by number of 64k pages
        MUL     BX
        MOV     TotalSizeHi,AX ; Set total size
        MOV     TotalSizeLo,0 ; Size is always a multiple of 64k
GPCExit:
        CALL    TurnVppOFF    ; Turn off Vpp

        RET
GetPhyChar ENDP
```

Listing 1—*Page wrap-around is detected by leaving the first device encountered in READ_ID mode. When a preexisting ID is found, the program knows to stop looping.*

By issuing the IOCTL read command, FEFS.SYS obtained the entry point to the low-level driver, which has been named FlashEntry. The FlashEntry procedure is to the FFS what the Interrupt procedure is to an MS-DOS device driver. It determines a command's validity before dispatching. FlashEntry handles calls from FEFS.SYS and applications that communicate directly with the flash memory, such as a formatter.

When writing the procedures to handle the FEFS.SYS commands, it is important to follow the entry and exit protocols. This is analogous to interrupt routines expecting parameters and status information to be passed within certain registers. Take for example the procedure ReadLogBlock that reads a block of data from the flash memory SSD into a buffer. FFS is informed by MS-DOS that the destination buffer is located at ES:BX. The CX register contains the number of bytes to read. A 32-bit pointer into flash memory is supplied by DI:DX. The unit number is passed in the AL register. Upon return from this procedure, the carry flag is expected to have the status of the operation, whether it was successful or unsuccessful.

There are thirteen procedures, including ReadLogBlock, that are defined by the Microsoft FFS technical specification. To more fully comprehend the file system, a brief discussion of each is helpful:

GetMediaCheck—Determines media status (same, changed, missing, unknown).

ReadLogBlock—Read a block of data at a logical address (which must be converted to a physical address).

WriteLogBlock—Writes a block of data from a buffer to the flash memory SSD.

CheckWrite—Checks a block of data for writability. If a byte is already programmed, but still has "one" bits, that byte can be rewritten to change the "ones" to a zero. This is useful for modifying the status byte.

EraseSSD—Erases data on a selected unit.

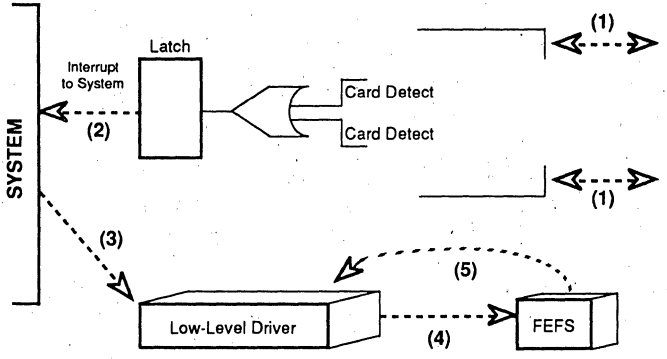
GetMediaInfo—Returns the description (type and size) of the installed media.

```
; Block Device Header use to set up drive letters with MS-DOS

Blockheader   DD   -1                  ; Last device in file
              DW   0000000000000000B  ; Block device
              DW   Strategy
              DW   BlockInt
bBlkUnits     DB   8 DUP (0)           ; Initialized in BlockInt

; These structures are passed back to MS-DOS to simulate
;  BIOS Parameter Block values:

BPB1          BPB  <512,1,1,2,128,1024,0F8H,2> ; Dummy values
BPB2          BPB  <512,1,1,2,128,1024,0F8H,2> ; Dummy values
```

Listing 2—*To reserve an MS-DOS drive letter, a different header must be used during driver installation. The new header acts like a block device, rather than a character device.*

FirstFree—Finds the first non-FFh byte from the end of the SSD and, as a result, finds the first available space.

SetMediaCB—Registers a procedure as the call-back procedure which is called by the LLD when a flash memory card is inserted or removed



(1) - Card is Removed or Reinserted
(2) - System is Notified of Interrupt
(3) - System Vectors to Interrupt Routine in LLD
(4) - LLD Notifies FEFS Through Callback Interface that Media Has Changed
(5) - FEFS Requests LLD to Perform a Media Interrogation

Figure 6—*The Call-back Interface supports changeable media in a system using Flash Cards.*

(see the discussion on Removable Media).

`GetMediaCB`—Returns the present media call-back procedure pointer. If zero is returned, no media call-back procedure exists.

`WriteLogByte`—Write a byte of data.

`ReadLogByte`—Read a byte of data.

`SetEraseCB`—Registers a procedure address as the call-back procedure called while erasing an SSD. It is used to display progress information.

`GetEraseCB`—Gets the address of the procedure called while erasing the SSD. A zero value returned indicates no erase call-back procedure has been registered.

```
BOOT_RCD        STRUC           ; Flash File System Boot Record
                                ;  (and root dir) copied in
                                ;  with formatter.
wStdID          DW 0F1A5H       ; Flash Media ID (Spells FLASH)
wUniqueIDLo     DW ?            ; Uniques ID for this Flash Memory
wUniqueIDHi     DW ?
WWriteVersion   DW ?            ; Flash version required to write
wReadVersion    DW ?            ; Flash version required to read
bPointerSize    DB 0            ; Num of bits in link list ptrs
pRoot           DB 3 DUP (0) ; Pointer to root directory
bVolumeLabel    DB 11 DUP (0FFH) ; Volume label
wEraseCyclesLo  DW 0FFFFH       ; Num times device erased (low)
wEraseCyclesHi  DW 0FFFFH       ; Num times device erased (high)

bManfID         DB ?            ; Manufacturer's ID
bDeviceID       DB ?            ; Device ID
wDeviceSize     DW ?            ; # of 64k pages in each device
wNumDevices     DW ?            ; # dev in SIMM or memory card
wTotalSizeLo    DW ?            ; Size of media (low word)
wTotalSizeHi    DW ?            ; Size of media (high word)
wTotalPages     DW ?            ; Total number of 64k pages
bIntelID        DB '1.0'        ; Indicates Intel format

pSibling        DB 3 DUP (0FFH) ; Start offset of data area
bRootName       DB 'ROOT    '; First entry must have this name
bRootExt        DB '   '
bStatus         DB 0FBH         ; Directory entry
pPrimary        DB 3 DUP (0FFH)
pSecondary      DB 3 DUP (0FFH)
bAttributes     DB 0FFH
wTime           DW ?
wDate           DW ?

BOOT_RCD        ENDS
```

Listing 3—*Any formatting utility must be hardware dependent. This listing shows the boot record copied into the SSD at the beginning of the FFS partition.*

SETTING UP THE PAGE NUMBER

During initialization, our software calculated the total flash memory available, and this information is passed back to `FEFS.SYS` (which knows nothing about the actual hardware used to access this memory). Because a page-memory scheme is used, the linear address supplied in DI:DX must be converted to a page number and offset within that page. A procedure, called `SetupRW`, performs this conversion and sets the hardware accordingly. In reality, if you are using a 64K-byte page size, this conversion is very simple: [DI] contains the page number and [DX] contains the offset. This is only tricky if you have designed your hardware with a smaller page size. In this case you would take DI and DX, move them into DX and AX, respectively, and divide DX:AX by the page size. AX would now contain the page number and DX the offset.

REMOVABLE MEDIA

If you have designed your page-memory board for the Intel Flash Memory card instead of SIMMs, you can include support for media change. The PCMCIA specifies the use of card-detect pins on the memory card. These pins can be wired to a latch that is set whenever the card is removed or inserted. This card-change line can be read before every access or can be tied to an interrupt line to allow instant acknowledgement of the card change. For both cases, Microsoft has provided a call-back interface.

A call-back interface allows the LLD to call a procedure internal to `FEFS.SYS` when the media changes. This function is similar to a software interrupt except it is private and cannot be accessed by other software. At initialization, `FEFS.SYS` requests the LLD to perform a "Set Media Call-back." In this procedure, the LLD stores the value in ES:BX which has been set up by `FEFS.SYS` as a pointer

to its internal media status call-back procedure. Now when the memory card is changed, the low-level driver, initiated by polling or interrupt, calls this procedure to let the FFS know that it needs to perform a media interrogation (Figure 6). This is analogous to removing a floppy disk in that MS-DOS must read the BPB and related information.

FORMATTING

A standard format utility is not incorporated in Microsoft's FFS because a formatter is hardware dependent. As such, a formatter was written that communicates directly with the LLD through `FlashEntry`. The formatter starts off by using the `EraseSSD` procedure. After erasing the SSD, a boot record (Listing 3) is copied to the SSD at the beginning of the FFS partition. During initialization, `wStdID` in the boot record is read to determine if the flash memory is formatted for the flash file system (notice that "F1A5h" spells Flash). If the SSD is FFS-compatible, the information in the remainder of the boot record describes the characteristics of the flash memory as well as the starting point for the FFS data area.

DEBUGGING

DOS Debug can be used for simple debugging purposes such as reading and writing I/O ports to test basic hardware functionality. Device drivers are difficult to debug because they are loaded during the DOS boot process. However, System Debugger by Sandpaper Software works well for this purpose. To use System Debugger to debug this device driver, the system must be set up so that a warm reboot may be taken at any time. After the debugger is loaded, breakpoints are set and a program called `BOOT.COM` is executed. This reboots the system without removing the debugger. When the breakpoint is reached, the debugger pops up and displays the information you need. One thing to note is that this debugger is designed for '386 systems because it uses the CPU's internal break registers and extended memory.

This flash file system project will provide a great education into the world of device drivers. Some of the concepts may seem a little tricky at first, but all of a sudden, a light bulb will go on, and you will be impressed by the functionality of this unique file system. Furthermore, once the project is complete, you will be impressed by the performance of the flash memory solid-state disk. This technology is rapidly gaining market acceptance and you have become a part of the secondary storage revolution. ✣

Markus Levy is an application engineer at Intel Corporation in Folsom, California, and holds a B.S.E.E. from California State University. His specialties include software and hardware implementations of solid-state disks in portable computers.